On universal classes of extremely random constant time hash functions and their time-space tradeoff

Alan Siegel
Courant Institute
New York University
New York, NY 10012

Abstract

A family of functions F that map $[0,n] \mapsto [0,n]$, is said to be h-wise independent if any h points in $[0,n]$ have an image, for randomly selected $f \in F$, that is uniformly distributed. This paper gives both probabilistic and explicit randomized constructions of n^ϵ-wise independent functions, $\epsilon < 1$, that can be evaluated in constant time for the standard random access model of computation. Simple extensions give comparable behavior for larger domains. As a consequence, many probabilistic algorithms can for the first time be shown to achieve their expected asymptotic performance for a feasible model of computation.

This paper also establishes a tight tradeoff in the number of random seeds that must be precomputed for a random function that runs in time T and is h-wise independent.

Categories and Subject Descriptors: E.2 [**Data Storage Representation**]: Hash-table representation; F.1.2 [**Modes of Computation**]: Probabilistic Computation; F2.3 [**Tradepffs among Computational Measures**]; F.2.1 [**Computation in finite fields**]; G.3 [**Probability and Statistics**]: Random number generation.

General terms: Algorithms, Theory.

Additional Key Words and Phrases: Hash functions, universal hash functions, hashing, limited independence, storage-time tradeoff.

The work of the author was supported in part by ONR grant N00014-85-K-0046, NSF grants CCR-8906949, CCR-8902221, and CCR-9204202.

1. Introduction

Many probabilistic algorithms and data structures have been proven to work well when fully random functions are used as unit time subroutines. For example, in the case of uniform hashing and double hashing, the expected cost for inserting the $(\alpha n+1)$-st item into a table of size n has been shown to be $\frac{1}{1-\alpha}+o(1)$ probes [6] and [7]. Moreover, this cost has been shown to be optimal for that genre of data access [24]. Yet the significance of these performance bounds for real computation is by no means clear. The difficulty is that they have been proven for hash functions that are assumed to be fully random. If, for example, we wish to hash data from $[1, n^2]$ into $[1, n]$, then there are n^{n^2} different functions that can perform such a mapping, and the program length of such a function must be about $n^2 \log n$ bits on the average. Such functions are much larger than the hash table they are intended to service.

On the other hand, results based upon full randomness sometimes translate into average case performance guarantees for real computation. In the case of double hashing, for example, which requires $2 \log n$ random bits per hash key, we may take these bits to be any fixed portion of the key itself, provided it is at least $2 \log n$ bits long. Then the probabilistic bound holds as an average case analysis. Just what to do for smaller number ranges is less clear. For uniform hashing, which requires additional randomness, the question of how to interpret a probabilistic upper bound on performance is even more problematic. Yet even where such average case results are meaningful, we would rather establish randomized performance bounds–which hold, on average, for any set of data–instead of a bound that cannot be applied to any fixed instance of data.

It is also worth noting that there are different kinds of probabilistic algorithms. Some require streams of disposable random bits, and a successful computation can be verified without their retention. In this case, there may be little need to store the random choices apart from concerns about reproducing the exact computation, and no need to have high speed random access for such data. Other algorithms may require

only a moderate number of random bits, which can be readily stored and accessed. For these applications, it may be sufficient to postulate the existence of a random bit source. Precomputed random strings might be streamed sequentially from secondary storage, and it is even conceivable that a source of quantum mechanical uncertainty could be used to generate the bits on the fly, provided its capacity and degree of randomness were adequate for the task at hand.

Other kinds of computations are based upon random functions, where some domain is mapped into a range according to, say, a uniform distribution, and the mapping of specific values must be recalled at periodic instances. This form of computation is less forgiving since the random decisions must be recalled. If the computation is dominated by such calculations, then the speed of the calculations may be important as well. In hashing, for example, random functions are used to locate items in a search table for subsequent retrieval. Ideally, the mapping of an item to a probe location should be done in constant time. Additional considerations include the storage allocated for the hash computation and the load or fraction of storage that can be occupied by data before the search performance becomes unacceptable. For large scale parallel computation, a random function might be shared by a large number of processors, and its program size, therefore, might be required to comprise a negligible percentage of local memory; it might also be required to exhibit high degrees of randomness, and to have a long expected lifetime before probabilistic events occur that require its replacement (with new random seeds).

Carter and Wegman introduced *universal hash functions* [3] and thereby provided a theoretical framework to formalize methods that exploit actual hash functions exhibiting fixed degrees of freedom. Related works [22], [11] have sometimes required a little more limited randomness, which is usually formalized along the following lines:

Definition 1.

A family of hash functions F with domain D and range R is (h, μ)-*wise independent*

if $\forall\ y_1, \ldots y_h \in R, \forall$ distinct $x_1, x_2, \ldots x_h \in D$:

$$|\{f \in F : f(x_i) = y_i,\ i = 1, 2, \ldots, h\}| \leq \mu \frac{|F|}{|R|^h}.$$

Thus the distribution of a random $f \in F$ on any h points is nearly uniform, and (h, μ)-wise independence implies (j, μ)-wise independence for $j < h$. For expositional simplicity, we will frequently suppress the μ parameter and simply refer to (h)-wise independence.

The limited randomness provided by such classes is frequently sufficient to achieve an expected performance for many randomized algorithms that is equivalent to the use of fully random hash functions. For example, recent randomized routing schemes for size n Omega networks have been proven to give optimal expected performance (up to constant factors), given a random $\Omega(\log n)$-wise independent hash function ([5], [13]). The hash functions used to date have typically been polynomials of degree $\beta \log n$ defined on finite fields.

In particular, Carter and Wegman exhibited the universal classes of (h)-wise independent hash functions that map $[0, m-1] \mapsto [0, n-1]$:

$$F_{(h)} = \{f \mid f(x) = (\sum_{0 \leq j < h} a_j x^j \mod p) \mod n,\ a_j \in [0, p-1]\}, \qquad (1)$$

where $p \geq m$ is prime. They showed that if, for any set $S \subset [0, p-1]$, a hash function is randomly selected from $F_{(h)}$ (independent of S), to bucket hash S into the n buckets $[0, n-1]$, then the sum of the expected j-th moments of the bucket populations is essentially the same as that resulting from fully random functions, for $j \leq h$. For bucket hashing with separate chaining, the second moment of the expected chain lengths (i.e. bucket populations) determines the expected retrieval time, whence pairwise independence guarantees optimal expected performance.

In the case of randomized routing on n-node bounded degree graphs, the $O(\log n)$ cost for each memory reference hashed by a function from $F_{(8 \log n)}$ is readily subsumed by the $\Omega(\log n)$ delay in routing the data ([5], [13]). Recently, $O(\log n)$-wise independent hash functions have also been shown to give optimal expected probe performance for

double hashing ([16]). But this efficiency is only in terms of probe counts; the cost to compute a single hash address is $c \log n$, given the hash functions developed to date. Thus the formal results of [16] are that dictionaries can be accessed with $O(\log n)$ computations per data operation, which is hardly surprising. For PRAM emulation on Omega networks (c.f. [5], [13), there would seem to be limited opportunity to exploit pipelining to mask latency for read intensive algorithms, as long as each address computation requires the evaluation of a polynomial having $\log n$ degree. Optimal speedup would appear to be beyond reach, even in theory.

Accordingly, it is reasonable to ask,

Is there an inherent log n penalty for computing such hash functions, or can we do better?

This paper shows how to trade the time complexity of (h)-wise independent hash functions for the number of random bits provided to it, and gives a mechanism for computing (n^{δ})-wise independent functions in $O(1)$ time from n^{ϵ} random words, for any fixed $\epsilon < 1$, and suitably fixed δ depending on ϵ and the word size of the domain. More precisely, the tradeoff is between the evaluation time for function evaluation and the workspace plus precomputation that is needed to provide a pool of random words. The actual number of random seeds required for the computation is $\theta(h)$, which is optimal.

Moreover, we establish a tight T-S-h tradeoff among the requisite number of probes T to a pool of S random seeds and the amount of independence h exhibited by the family of random hash so constructed. The actual construction simply combines the probed data values with the "Exclusive-Or" operator and uses twice the number of probes proved necessary by the lower bound. While the question of how to compute these probe sequences effectively is still open, we show that constant time random families with very high independence are programmable, for a constant that is exponential in the time predicted by a nonuniform model of computation.

An immediate consequence of these constructions is that double hashing using these universal functions has (constant factor) optimal performance in time, for load factors bounded below 1. Another consequence is that a T-time PRAM algorithm for $n \log n$ processors (and n^k memory) can be emulated on an n-processor machine interconnected by an $n \times \log n$ Omega network with a multiplicative penalty for total (non-switching) work that, with high probability, is only $O(1)$; optimal speedup is achieved.

The paper is organized as follows. Section 2 presents three random function constructions, from probabilistic (i.e., nonconstructive) but extremely efficient, to programmable (with code). They all run in constant time and are effectively (n^δ)-wise independent, for different, suitably small fixed $\delta > 0$. Section 3 gives a lower bound to show that the first construction is optimal, in terms of the number of random words that are used per function evaluation. Section 4 gives a few applications while Section 5 presents the conclusions and the main open question concerning these hash functions.

2. The hash function

The motivating question leading to our constructions is based on the following simple observation. Given h random elements from domain $[0, m-1]$, these coefficients can be used as in equation (1) to construct an (h)-wise independent hash function. Evidently, evaluation requires $O(h)$ time. If m random elements are provided, then table lookup gives an $O(1)$ time function. What sort of random functions can be constructed from n^ϵ random seeds?

For the purposes of this paper, we might view the physical storage as size n within a virtual address space of n^k, and take n^ϵ to be an acceptable portion of space to allocate for random function[†] computation. Our underlying model of computation is the Random Access Machine where both memory access and the basic arithmetic and logic operations can be executed on words in unit time (c.f.[1]).

[†]The performance, on the other hand, only gets better if more space is available for the hashing operation.

We temporarily suppress the issue of program size and construct a family of fast highly independent hash functions that map $[0, n^k - 1]$ onto $[0, n^k - 1]$ and use n^c words of random input. We are also suppressing the issue of domain size. The reason this can be done is well known: a simple linear congruence hash function can be used to map any fixed set of $s \leq n$ elements from a large domain $D = [0, m]$ into the small domain $[0, n^k]$, so that the probability no collisions occur is at least $1 - \frac{1}{n^{k-2}}$. Such mappings can be pieced together from techniques in [3], [9] and [4]. The details of this construction are in Appendix 1.

Consequently, this space reducing prehashing step has only a minimal impact on the performance of the resulting hash functions. We may formalize this fact as follows.

Definition 2.

A family of hash functions F with domain D and range R is *r-practical* (h, μ)*-wise independent* if for any subset $S \in D$, with $|S| \leq n$, $\exists \bar{F} \subset F : |F - \bar{F}| \leq |F|/|R|^r$ and $\forall\ y_1, \ldots y_h \in R, \forall$ distinct $x_1, x_2, \ldots x_h \in S$:

$$\frac{|\bar{F}|}{\mu |R|^h} \leq |\{f \in \bar{F} : h(x_i) = y_i, i = 1, 2, \ldots, h\}| \leq \frac{\mu |\bar{F}|}{|R|^h}.$$

The real point of this definition is to quantify the performance of good hash functions that are constructed by a randomized algorithm, which might include a prehashing step that randomly selects a prime $p \approx n^k$. For most applications, it suffices to have almost all hash functions exhibit the collective randomness that is desired. If a randomly selected function is from a poorly behaved subset, we can, depending on the underlying process at hand, attribute a cost of n^l for using it, where, say, $l < r$. Then the expected performance penalty for these bad functions is a negligible $O(\frac{n^l}{|R|^r})$. The reason that our hash functions are defined on $[0, n^k - 1]$ for fixed but unspecified k is to expose the tradeoffs in computational resources and residual errors such as that caused by the prehashing step that contracts our domain to a polynomial size. The bound on the distribution for $\bar{F}$ is stated in a two-sided form to facilitate inclusion-exclusion calculations. Such formulations hold for most families of universal hash functions defined

to date, although the original definitions of (h)-wise independence have not required it. As for the factor μ, our basic constructions satisfy the criterion for $\mu = 1$, as do the degree $h-1$ polynomials taken mod p.

The crux of the problem, then, is to construct (h)-wise independent maps from $[0, p-1]$ onto $[0, p-1]$, for a fixed prime $p > n^k$ (or where p is a power of 2). If the ultimate intended range is smaller, say $[0, n-1]$, then a final postprocessing that computes the outcome mod n will give the result with a small number of hash functions that skew onto $[0, p \bmod n]$ those data items provisionally mapped into $[p - n\lfloor \frac{p}{n} \rfloor, p-1]$. This final skewing increases in μ by a factor of $(1 + n/p)^h$, which is quite modest for, say, $p > hn^2$.

We shall restrict, for the moment, the problem to constructing fully $(h, 1)$-wise independent hash functions that map $[0, p-1]$ onto $[0, p-1]$, given an auxiliary pool of about n^ϵ random $(\log p)$-bit words, for some $\epsilon < 1$, and fixed prime $p \approx n^k$. Now any random hash function must have a mechanism that associates each element in $[0, p-1]$ with a few of these random words, as otherwise no random computation can result. If the association is deterministic, then it can be represented by a bipartite graph G on the vertex sets[†] $[0, p-1]$ and $[0, p^\epsilon - 1]$). Moreover, such a bipartite graph must associate at least l random numbers with each set of l elements from $[0, p-1]$, for $l \leq h$, as otherwise there are not enough degrees of freedom to achieve (h)-wise independence. According to Hall's Theorem, which is also known as the Marriage Theorem, this criterion is equivalent to every subset of h elements in $[0, p-1]$ having a matching in the graph with its neighbors, which comprise a small subset contained within the p^ϵ words. Suitable graphs are formalized as follows.

Definition 3.

Let a (p, ϵ, d, h)-*weak concentrator* be a bipartite graph on sets of vertices I (inputs) and O (outputs), where $|I| = p$, $|O| = p^\epsilon$, and the following hold. Each input has

[†]Of course the graph could be defined with the first vertex set restricted to just $[0, n^k - 1]$.

outdegree d. Any set of h inputs has edges that achieve a matching with some h outputs.

Our next observation is that these graphs can be constructed with very small outdegree d.

Lemma 1. For $r \geq 0$, $d = 2 + \frac{(r+1)}{\epsilon}$ and $h \leq \frac{p^{\frac{\epsilon r+\epsilon^2}{r+1}}}{e^{\frac{2\epsilon}{(r+1)}}}$, (p, ϵ, d, h)-weak concentrators exist.

Proof: We use the probabilistic method (c.f. [15]) to estimate the probability that a randomly constructed graph will fail to meet the matching criterion. The construction assigns, to each node in $I = [0, p-1]$, edges to d distinct random nodes in $O = [0, p^\epsilon]$. Thus a matching is guaranteed for subsets of d or fewer vertices in I. For larger aggregates of size at most h, Hall's Theorem says that there will always be a matching if and only if each subset of $j \leq h$ vertices in I has edges to at least j vertices in O. The probability that some such aggregate fails to have a matching is less that the expected number of such subsets that fail Hall's criterion, which is the expected number of subsets in I of size j whose neighbors lie within some subset of $j-1$ vertices in O. In particular, the probability of a failure is overestimated by the expected number of pairs $(S \subset I,\ T \subset O)$, where $|S| = j$, $|T| = j-1$, and all jd edges from S have destinations within T, for $d < j \leq h$. Evidently, the probability that the jd edges are so selected, for any fixed (S, T), is $\left(\frac{\binom{j-1}{d}}{\binom{p^\epsilon}{d}}\right)^j < (\frac{j-1}{p^\epsilon})^{jd}$, and the number of candidate (S,T) pairs is just $\binom{p}{j}\binom{p^\epsilon}{j-1}$.

Following this prescription, we can estimate that the probability a randomly generated (p, ϵ, d, d)-weak concentrator fails to be an (n, ϵ, d, h)-weak concentrator by summing over all relevant pairs of subsets to get:

$$\begin{aligned}
\Pr\{\text{failure}\} &< \sum_{d<j\le h} \binom{p}{j}\binom{p^\epsilon}{j-1}\left(\frac{\binom{j-1}{d}}{\binom{p^\epsilon}{d}}\right)^j \\
&< \sum_{d<j\le h} \binom{p}{j}\binom{p^\epsilon}{j-1}\left(\frac{j-1}{p^\epsilon}\right)^{jd} \\
&< \sum \frac{p^{j+j\epsilon} j^{2j+1+j(r+1)/\epsilon}}{j!j!p^{2j\epsilon+(r+1)j+\epsilon}} \\
&<^\dagger_s \sum \frac{e^{2j}}{j^{2j+1}} \frac{p^{j+j\epsilon} j^{2j+1+(r+1)j/\epsilon}}{p^{2j\epsilon+(r+1)j+\epsilon}} \\
&< p^{-\epsilon} \sum_{j\le h} (e^2 j^{(r+1)/\epsilon}/p^{\epsilon+r})^j \\
&< p^{-\epsilon} \sum_{j\le h} (e^2 h^{(r+1)/\epsilon}/p^{\epsilon+r})^j < p^{-\epsilon} \sum_{j\le h} 1^j < \frac{h}{p^\epsilon} < 1.
\end{aligned}$$

Since the probability is less than 1 that a randomly constructed graph fails to be a (p,ϵ,d,h)-*weak concentrator*, it follows that such a construction will succeed with positive probability and hence these graphs do indeed exist. ∎

We have, as yet, no hash function; but each element, at least, is now associated with a few random values. The obvious use for these values is as coefficients of a hashing polynomial. By increasing the number of random values used in this calculation, we can turn a weak concentrator into a calculation procedure for fully (h)-wise independent hash functions.

Let G be a (p,ϵ,d,h)-weak concentrator. For each input i in G, let i's d neighbors in G be stored in the set $Adj(i)$. Let M_m be a $p^\epsilon \times d$ array of words in $[0,p-1]$, whose concatenated contents is $m \in [0,p-1]^{p^\epsilon d}$, for some prime $p > n^k$.

Define the random hash function

$$f_m^G(i) = \sum_{j\in Adj(i), 0\le l<d} M_m(j,l) i^l \pmod p.$$

Thus a computing $f_m^G(i)$ requires d^2 additions and d multiplications plus a comparable number of modular divisions. The result turns out to be (h)-wise independent.

---

†We are using a simple version of Stirling's Formula: $\sqrt{j} j^j e^{-j} < j!$, for $j > 0$. Subsequent applications will be denoted by the annotated inequality sign $<_s$.

Lemma 2. Let G be a (p, ϵ, d, h)-weak concentrator. Then $\{f_m^G\}_{m \in [0,p-1]^{dp^\epsilon}}$ is an $(h, 1)$-wise independent family of hash functions mapping $[0, p-1] \mapsto [0, p-1]$.

Proof: It is not difficult to see that we need only establish the linear independence of the systems of equations that constrain the m values to yield arbitrarily specified values for f, on any h inputs. This is because such a system has h constraints in dp^ϵ unknowns. If the system enjoys linear independence, then the null space has dimension $dp^\epsilon - h$, and each set of h values will be attained for $p^{dp^\epsilon - h}$ of the p^{dp^ϵ} sets of random words. So suppose that specifying values for some input set I_0, where $|I_0| \leq h$, induces a minimal dependent linear system. That is, a linear combination of the rows in the linear system with row indices in I_0 sums to the zero vector, and no row has a coefficient of zero in the linear combination. Now I_0 sources $d|I_0|$ edges, which reach at least $|I_0|$ outputs, so there must be an output $y_0 \in O$ having exactly q edges that originate in I_0, for some q where $0 < q \leq d$. Consider the linear subsystem with rows indexed by $I_1 = \{i \in I_0 : y_0 \in Adj(i)\}$. By definition of y_0, $1 \leq |I_1| = q \leq d$. The subsystem restricted to the variables $M(y_0, k)$, where $k = 0, 1, \ldots d-1$ has coefficients that are the Vandermonde submatrix

$$\begin{pmatrix} 1 & i_1 & i_1^2 & \ldots & i_1^{d-1} \\ 1 & i_2 & i_2^2 & \ldots & i_2^{d-1} \\ \vdots & \vdots & \vdots & & \vdots \\ 1 & i_q & i_q^2 & \ldots & i_q^{d-1} \end{pmatrix}$$

where I_1 comprises the distinct rows $(i_1, i_2, \ldots, i_q)$. As is well known and easily verified, the determinate of the matrix (in the case $q = d-1$) is $\prod_{0<j<k\leq q}(i_k - i_j)$, which shows that such a subsystem cannot be linearly dependent because no two rows are the same. Since none of other rows with indices in I_0 have any of the variables $M(y_0, 0), M(y_0, 1), \ldots, M(y_0, d-1)$ present, the assumption that the system is minimal and dependent is contradicted. ∎

So far, we have a probabilistic fast hashing procedure that is (h)-wise independent, uses dp^ϵ random words of $\log p$ bits, and requires d^2 additions and d multiplications per evaluation. The construction gives a generic transformation from a graph rich in

matchings to a family of highly random functions. We now give a more efficient construction that uses better random graph properties, with a constant factor degradation in the independence (or outdegree d), but where only one random value is stored per output destination. The construction finds a sparse bipartite graph where every h rows of its adjacency matrix is linearly independent. In fact, the linear independence holds for $GF(2)$, the field of integers mod 2.

Definition 4.

Let an (n, ϵ, d, h)-*weakly triangular* graph be a bipartite graph on sets of vertices I (inputs) and O (outputs), where $|I| = n$, $|O| = n^\epsilon$, and the following hold. Each node in I has an outdegree of at most d. The $|I| \times |O|$ adjacency matrix of the graph, when restricted to any h input rows, and all $|O|$ columns, can be be permuted into an upper triangular form with nonzero diagonal: suitable row and column permutations transform the $h \times n^\epsilon$ submatrix S so that $S(i,l) = 0$, if $i > l$ and $S(i,i) = 1$, for $i = 1, 2, \ldots, h$.

Of course any (n, ϵ, d, h)-weakly triangular graph is also (n, ϵ, d, j)-weakly triangular, for $j < h$.

Lemma 3. Let G be (n, ϵ, d, h)-weakly triangular. For each input i in G, let i's neighbors in G be stored in the set $Adj(i)$. Let M_m be an array of n^ϵ words in $[0, n-1]$, with concatenated contents $m \in [0, n-1]^{n^\epsilon}$, where n is a power of 2. Define the random hash function

$$f_m^G(i) = \mathbf{XOR}_{j \in Adj(i)} M_m(j),$$

where **XOR** is the bitwise "Exclusive-Or" function (or any other commutative group operation such as modular addition). Then $\{f_m^G\}_{m \in [0,n-1]^{n^\epsilon}}$ is an $(h, 1)$-wise independent family of hash functions mapping $[0, n-1] \mapsto [0, n-1]$.

Proof: As in Lemma 2, we need only show that a minimal dependent set of row vectors must comprise $h+1$ or more vectors. But this is immediate, since the system is

easily solved by the back substitution step of Gaussian elimination. Given j equations, identify a variable that appears only once and first solve the $j-1$ equations that are independent of it. Then the last equation is readily solved under our commutative group operation such as the "Exclusive-Or". Since the number of solutions to j such equations in l unknowns is n^{l-j}, the $(h,1)$-wise independence is ensured. ∎

We now show that some random graphs are weakly triangular. Later others will also be shown to have this property.

Definition 5.

Let an (n,ϵ,d,h)-*weak expander* be a bipartite graph on sets of vertices I (inputs) and O (outputs), where $|I|=n$, and $|O|=n^{\epsilon}$, and the following hold. Each input has an outdegree bounded by d. Any set of j inputs, for $1<j\le h$, has edges to at least $\lfloor jd/2\rfloor+1$ different outputs.

Lemma 4. An (n,ϵ,d,h)-*weak expander* is (n,ϵ,d,h)-weakly triangular.

Proof: For $j>1$, any $j\le h$ rows have at least $\lfloor jd/2\rfloor+1$ different output variables, whence at least one such variable will appear in exactly one of the rows. Permuting this variable and row into location $S(1,1)$ leaves $j-1$ rows with the same property, whence recursion completes the construction. ∎

Lemma 5. For $\epsilon > \frac{2}{d}+\frac{1+\log d+\log h}{\log n}$, (n,ϵ,d,h)-weak expanders exist.

Proof: Proceeding as in Lemma 3 gives, for our previous random construction:

$$\begin{aligned}
\text{Prob\{failure\}} &< \sum_{1<j\le h}\binom{n}{j}\binom{n^{\epsilon}}{jd/2}\left(\frac{\binom{jd/2}{d}}{\binom{n^{\epsilon}}{d}}\right)^{j}\\
&<_s \sum_{j\le h}(n^{1-\epsilon d/2}(jde/2)^{d/2})^{j}/j!\\
&< \sum_{j\le h}(e^{-(1+\log d+\log h)d/2}(hde/2)^{d/2})^{j}/j!\\
&< \sum_{j\le h}(1/2)^{jd/2}/j! < 1. \quad ∎
\end{aligned}$$

Combining Lemmas 3 and 4 with Lemma 5 where $\binom{n}{j}$ is replaced by $\binom{n^k}{j}$ gives the following.

Theorem 1. For $\epsilon \geq \delta + \frac{2k}{d} + \frac{1+\log d}{\log n}$, there are fixed programs that, for each input from $[0, n^k - 1]$, "Exclusive-Or" together d words from a pool of n^ϵ random $(k \log n)$-bit words to compute an $(n^\delta, 1)$-wise independent hash function mapping $[0, n^k - 1] \mapsto [0, n^k - 1]$, where n is a power of 2. ∎

It is worth observing that the collections of n^ϵ random numbers used in Theorem 1 will yield a family of (h)-wise independent hash functions if they are selected from a family of (dh)-wise independent pools. Thus each pool can be precomputed from dh random seeds. Consequently the space-time tradeoff, for families of fast highly independent hash functions, is not a function of the number of random bits that must be specified (which is essentially its Kolmogorov complexity) but is really matter of intrinsic storage requirements.

This second construction gives a more efficient family of hash functions, and again provides a generic procedure that turns a good graph into a family of hash functions. It does not quite supersede the first construction because there are no known explicit graphs of either type. Should a short (deterministic or effective probabilistic) algorithm be found, which builds weak concentrators where an input's adjacency list can be generated in constant time, then fast highly independent hash functions will follow. Similarly, effective procedures for constructing weak expanders will yield even better hash functions.

The difficulty with the constructions presented so far is that the random graphs G require a huge description for their dn edges. Moreover, the problem of finding such a graph seems to be quite difficult. Fortunately, Cartesian products can be used to attain compact representations of less efficient hash functions, where we forgo some randomness, and increase our $O(1)$ operation count to an exponentially larger constant. This section closes with a simple analysis of what can be done to achieve fast highly independent hash functions that are less efficient than the constructions given so far, but which nevertheless run in constant time and are spatially compact. These variations

can be applied to either formulation, but we shall restrict our attention, for the most part to the latter, since they appear to be more efficient.

Definition 6.

Let the *Cartesian product* $A \times B$ of two bipartite graphs $A = (I, O, E)$ and $B = (U, V, F)$ be the graph $C = (W, X, H)$, with input vertex set $W = I \times U$, output set $X = O \times V$, and edge set H, which contains the edge from $(i, u) \in W$ to $(o, v) \in X$ if and only if $edge(i, o) \in E$ and $edge(u, v) \in F$.

Lemma 6. Let G_1 be (n, ϵ, d, h)-weakly triangular, and G_2 be (m, ϵ, c, h)-weakly triangular. Then the Cartesian product $G_1 \times G_2$ is (mn, ϵ, cd, h)-weakly triangular.

Proof: We need only verify the weak triangularity property for $G_1 \times G_2$. Let $x_1, x_2, \ldots x_l$ be an arbitrary set of $l \leq h$ distinct inputs in $[0, n-1] \times [0, m-1]$. Let x_i have the Cartesian product representation $x_i = (z(i,1), z(i,2)), z(i,1) \in G_1$, $z(i,2) \in G_2$. The following procedure finds a permutation that is upper triangular for the x_i.

Initialize the permutation of rows and columns to $\emptyset$;

Mark all outputs from G_1 as free;

Assign $Z_1 \leftarrow \{z | z(i,1) = z$ for some $i \in [1, l]\}$. %By construction, Z_1 is nonempty.

repeat

Delete a $z_1 \in Z_1$ that has a free output o_1 that is not an output of any other $z \in Z_1$;

%This can be done because G_1 is weakly triangular.

Mark o_1 as not free;

Assign $I \leftarrow \{a | a \in [1, l]$ and $z(a, 1) = z_1\}$; %All such x_a agree in the G_1 coordinate.

Mark all outputs in G_2 as free;

Assign $Z_2 \leftarrow \{z | z(i, 2) = z$ for $i \in I\}$;

for each $i \in I$ **do**

Delete a $z_2 \in Z_2$ that has a free output o_2 that is not an output of another $z \in Z_2$;

Mark o_2 as not free;

Assign row (z_1, z_2) and column (o_1, o_2) to be next in the permutation

endfor

until Z_1 is empty;

return the permutation.

It is easy to see that the algorithm orders the nodes in an upper triangular order. ∎

In particular, if G is an $(n^\epsilon, \epsilon, d, h)$-weak expander, then the Cartesian product $G^{1/\epsilon}$ is $(n, \epsilon, d^{1/\epsilon}, h)$-weakly triangular.

Combining Lemmas 3,4,5, and 6 with suitable rescaling gives gives the following.

Theorem 2. For $\epsilon \geq \frac{2k}{d} + \frac{1+\log d+\log h}{\epsilon \log n}k$, there are fixed programs of size $O(\epsilon^2 d/k)n^\epsilon$ that, for each input from $[0, n^k - 1]$, "Exclusive-Or" together $d^{k/\epsilon}$ words from a pool of n^ϵ random $(k \log n)$-bit words to compute an $(h, 1)$-wise independent hash function mapping $[0, n^k - 1] \mapsto [0, n^k - 1]$, where n^ϵ is a power of 2.

Proof: We store an explicit $(n^\epsilon, \epsilon/k, d, h)$-weak expander G as part of the hash function; a value $i \in [0, n^k - 1]$ is hashed by computing the adjacency list for i in $G^{k/\epsilon}$, and applying the "Exclusive-Or" to the random words so probed as the neighbors of i.

It only remains to verify the existence of an $(n^\epsilon, \epsilon/k, d, h)$-weak expander for the parameters at hand. Computing as before gives,

$$\begin{aligned}
\text{Prob\{failure\}} &< \sum_{1<j\leq h} \binom{n^\epsilon}{j}\binom{n^{\epsilon^2/k}}{jd/2}\left(\frac{\binom{jd/2}{d}}{\binom{n^{\epsilon^2/k}}{d}}\right)^j \\
&<_s \sum_{j\leq h} (n^{\epsilon-\epsilon^2 d/2k}(jde/2)^{d/2})^j/j! \\
&< \sum_{j\leq h} (n^{-\epsilon(\frac{1+\log d+\log h}{\log n})\frac{k}{\epsilon}d/2k}(jde/2)^{d/2})^j/j! \\
&< \sum_{j\leq h} (e^{-(1+\log d+\log h)d/2}(hde/2)^{d/2})^j/j! \\
&< \sum_{j\leq h} (1/2)^{jd/2}/j! < 1. \quad ∎
\end{aligned}$$

The conditions of Theorem 2 can be simplified to yield, for example, an $(n^\delta, 1)$-wise independent family of hash functions on $[0, n^k - 1]$, when $\epsilon = \frac{2k}{d} + \sqrt{k(\delta + \frac{1+\log d}{\log n})}$. The functions can be evaluated in time $d^{k/\epsilon}$, and have a program size of $O(n^\epsilon)$. The point of this construction is that for fixed k and large, slowly growing h, $\epsilon = O(\frac{1}{d}) + o(1)$ as

$n \to \infty$.

For completeness, we state without proof the analogous composition formulation for the first construction.

Lemma 7. Let G be an $(n^\epsilon, \epsilon, d, h)$-weak concentrator. Then the Cartesian product $G^{1/\epsilon}$ is an $(n, \epsilon, d^{1/\epsilon}, h)$-weak concentrator. ∎

The families of hash functions have as yet been "demonstrated" only in a probabilistic sense; no explicit constructions have been given. Formally, (that is, up to constant factors) this distinction is moot. By increasing, slightly, the degrees of freedom in our probabilistic constructions, the same counting argument will ensure that with probability $1-1/n^r$, a randomly selected graph is a weak concentrator or expander. Accordingly, we may simply increase the size of the hash family by indexing it over all graphs satisfying the (modified) size and degree parameters of Lemma 5. The resulting randomized construction $F_M^{G(n,\epsilon,d)}$ is an explicit family of $O(1)$ time hash functions that is essentially (h)-wise independent, as characterized by Definition 2.

Lemma 8. For $\epsilon > \frac{2+r}{d} + \frac{1+\log d+\log h}{\log n}$, a random bipartite graph on $[0, n-1] \times [0, n^\epsilon - 1]$ with outdegree d is a (n, ϵ, d, h)-weak expander with probability exceeding $1 - \frac{1}{n^r}$.

Proof: We apply the random construction used for Lemma 5, but include the (algorithm) simplifying modification that each input vertex receives d edges selected at random with replacement:

$$\begin{aligned}
\text{Prob}\{\text{failure}\} &< \sum_{1<j\le h} \binom{n}{j}\binom{n^\epsilon}{jd/2}\left(\frac{jd/2}{n^\epsilon}\right)^{jd} \\
&<_s \sum_{2\le j\le h} (n^{1-\epsilon d/2}(jde/2)^{d/2})^j/j! \\
&< \frac{1}{n^r}\sum_{j\le h}(1/2)^{jd/2}/j! < n^{-r}. \quad ∎
\end{aligned}$$

Combining Lemmas 3,4, and 8, substituting n^k for n, and $n^{\epsilon/k}$ for n^ϵ gives the following characterization of the family $F_M^{G(n,\epsilon,d)}$.

Theorem 3. For $\epsilon \ge \frac{2k}{d} + \frac{kr}{d\epsilon} + k\frac{1+\log d+\log h}{\epsilon \log n}$, $F_M^{G(n,\epsilon,d)}$ is an explicit family of r-practical

$(h,1)$-wise independent hash functions mapping $[0,n^k-1] \mapsto [0,k-1]$. $F_M^{G(n,\epsilon,d)}$ has a program space of $\frac{d\epsilon^2}{k}n^\epsilon + O(1)$ $(\log n)$-bit words, and for each input from $[0,n^k-1]$, computes the "Exclusive-Or" of $d^{k/\epsilon}$ members in a pool of n^ϵ random $(k\log n)$-bit words. The requirement for ϵ is readily simplified to $\epsilon \geq \frac{2k}{d} + \sqrt{\frac{kr}{d} + k\frac{1+\log d+\log h}{\log n}}$.

Proof: The program for F_M^G is essentially an array A of dn^ϵ words belonging to $[0,n^{\epsilon^2/k}]$. The d edges emanating from vertex $i \in [0,n^\epsilon-1]$ of G are found in locations $A[l]$, for $di \leq l < (d+1)i$. Given $j \in [0,n^k-1]$, the $d^{k/\epsilon}$ locations among the n^ϵ random words are found by expanding the edges from vertex j of $G^{1/\epsilon}$ on the fly in $O(d^{k/\epsilon})$ time. The program for this expansion requires $O(1)$ space.

Thus it suffices to verify the existence of an $(n^\epsilon, \epsilon/k, d, h)$-weak expander for the parameters at hand. Computing as before gives,

$$\begin{aligned}
\text{Prob}\{\text{failure}\} &< \sum_{1<j\leq h} \binom{n^\epsilon}{j}\binom{n^{\epsilon^2/k}}{jd/2}\left(\frac{\binom{jd/2}{d}}{\binom{n^{\epsilon^2/k}}{d}}\right)^j \\
&<_s n^{-r} \sum_{1<j\leq h} (n^{\epsilon-\epsilon^2 d/2k}(jde/2)^{d/2})^j/j! \\
&< n^{-r} \sum_{1<j\leq h} (n^{-\epsilon(\frac{1+\log d+\log h}{\log n})\frac{k}{\epsilon}d/2k}(jde/2)^{d/2})^j/j! \\
&< n^{-r} \sum_{1<j\leq h} (e^{-(1+\log d+\log h)d/2}(hde/2)^{d/2})^j/j! \\
&< n^{-r} \sum_{1<j\leq h} (1/2)^{jd/2}/j! < n^{-r}. \quad \blacksquare
\end{aligned}$$

Here the bipartite graph is part of the random input, whereas before one good graph was shown to service the entire family of hash functions. Thus in the former case, an amortized randomized algorithm might require, upon rare occasion, new random seeds to attain a better family member for the current data, but the graph would last forever; for applications of Theorem 3, the new hash function candidate would include randomly selected edges for a new random graph among its random seeds.

For completeness, a rather crudely transparent iterative version of the algorithm is presented below.

function Random(i: in $[0,n^k-1]$): in $[0,n^k-1]$;

Global A: $n^\epsilon \times 1$ array of words in $[0,n^k-1]$;

Global G: $n^\epsilon \times d$ array of words in $[0, n^{\epsilon^2/k} - 1]$;

Local $l_1, l_2, \ldots, l_{k/\epsilon}$: in $[0, d-1]$;

Local $i_1, i_2, \ldots, i_{k/\epsilon}$: in $[0, n^\epsilon - 1]$;

Local j: in $[0, n^\epsilon - 1]$;

Local val: in $[0, n^k]$;

Assign $(i_1, i_2, \ldots, i_{k/\epsilon}) \leftarrow i$;

$val \leftarrow 0$;

for $l_1 \leftarrow 0$ **to** $d-1$ **do**

  for $l_2 \leftarrow 0$ **to** $d-1$ **do**

    $\vdots$

      for $l_{k/\epsilon} \leftarrow 0$ **to** $d-1$ **do**

        Assign $j \leftarrow (G[i_1, l_1], G[i_2, l_2], \ldots, G[i_{k/\epsilon}, l_{k/\epsilon}])$;

        $val \leftarrow (A[j]$ **XOR** $val)$

endallfors; % Altogether, $d^{k/\epsilon}$ **XOR**s take place.

return(val).

3. A lower bound

We now show that the size of our random word pool cannot be materially reduced without affecting the running time of the hash function. A family of (h, μ)-wise independent hash functions $F_M = \{f_m(x)\}_{m \in M}$ where $f_m : S \mapsto S$ will be modeled as follows. Each f_m is defined by the same algorithm, which inputs x and then reads d locations in an array $A[1..z]$, that contains z values belonging to S. Index m is the string of concatenated data contained in A. The algorithm can even be viewed as probabilistic since values found in A might be used with x in an adaptive search to determine which other array locations to access. These values and x are then used deterministically to compute the random function value in S. Let $n^{\underline{j}} = n(n-1)(n-2)\ldots(n-j+1)$.

Theorem 4. Let $F_M = \{f_m\}_{m \in M}$ denote a family of (h, μ)-wise independent hash functions mapping $S \mapsto S$, where $M \subset S^z$. Then the time complexity T to evaluate

$f \in F_M$ satisfies either $T \geq h$ or

$$z^{\frac{T}{}} > (h-2)^{\frac{T-1}{}}(|S| - \mu).$$

Proof: We may suppose that each computation of f examines d entries in the array A. We show that d satisfies the constraint for T. For each set ζ of $h-1$ locations in the z element array A, we partition M into $M^\zeta = \left\langle M_1^\zeta, M_2^\zeta, \ldots, M_{|S|^{h-1}}^\zeta \right\rangle$, where M_i^ζ is the set of strings in M that equal, on ζ, the i-th enumeration of a fixed ordering of S^{h-1}. Let $S(\zeta, i)$ be the set of domain elements $x \in S$ that, when computing $f_m(x)$ for $m \in M_i^\zeta$, have their d A-locations read from within ζ. Let

$$S_0(\zeta, i) = \begin{cases} S(\zeta, i), & \text{provided } |M_i^\zeta| > \mu|M|/|S|^h; \\ \emptyset & \text{if } |M_i^\zeta| \leq \mu|M|/|S|^h. \end{cases}$$

Given an $s \in S_0(\zeta, i)$, $f_m(s)$ will be computed by probing the same d-tuple of locations within ζ for all $m \in M_i^\zeta$.

There are $\binom{z}{h-1}$ subsets ζ, and each subset induces a partition of M indexed by the m-values restricted to ζ. Let $\Sigma = \sum_\zeta \sum_i |M_i^\zeta||S(\zeta, i)|$, and set $\Sigma_0 = \sum_\zeta \sum_i |M_i^\zeta||S_0(\zeta, i)|$. It follows that $|S_0(\zeta, i)| < h$ since otherwise there are h elements in S that hash to some tuple with probability exceeding $\mu/|S|^h$. Hence $\Sigma_0 \leq \sum_\zeta \sum_i (h-1)|M_i^\zeta| = \sum_\zeta (h-1)|M|$, whence

$$\Sigma_0 \leq (h-1)\binom{z}{h-1}|M|. \tag{2}$$

On the other hand, each m-string in M will be probed, for each $s \in S$, in d locations, which means that the pair (s, m) is counted exactly $\binom{z-d}{h-1-d}$ times in Σ. Hence

$$\Sigma = \binom{z-d}{h-1-d}|S||M|. \tag{3}$$

Finally, each $s \in S$ may be able to encounter each of the $|S|^d$ sequences of probe values within $\binom{z-d}{h-1-d}$ different ζ sets, which have $h-1-d$ unprobed locations that can have up to $|S|^{h-1-d}$ different assignments. That is, any $s \in S$ belongs to at most $\binom{z-d}{h-1-d}|S|^{h-1}$ different $S(\zeta, i)$. In view of this, we can count that $\Sigma - \Sigma_0 < \sum_\zeta \sum_i \frac{\mu|M|}{|S|^h}|S(\zeta, i)| \leq \frac{\mu|M|}{|S|^h}|S|\binom{z-d}{h-1-d}|S|^{h-1}$, whence

$$\Sigma - \Sigma_0 < \binom{z-d}{h-1-d}\mu|M|. \tag{4}$$

Combining equation (3) and inequality (4) gives

$$\Sigma_0 > \binom{z-d}{h-1-d}|M|(|S|-\mu). \tag{5}$$

Combining inequalities (2) and (5) gives

$$(h-1)\binom{z}{h-1} > \binom{z-d}{h-1-d}(|S|-\mu).$$

Eliminating common factors establishes that

$$\frac{z^{\underline{d}}}{(h-2)^{\underline{d-1}}} > |S|-\mu. \quad \blacksquare$$

Notice that when h random probes to the pool are allowed, the bound on d collapses to the empty requirement $z^{\underline{d}} > 0$. Of course h random numbers are necessary and sufficient, in this case. For $d < h$, at least $d = \frac{k+\frac{\log h}{\log n}-o(1)}{\epsilon}$ probes are needed per evaluation of an (h)-wise independent hash function that uses a database of $z = n^{\epsilon}$ random $k \log n$-bit words to map $[0, n^k - 1] \mapsto [0, n^k - 1]$. In view of Theorem 1, we conclude that

$$T = \Theta(k/\epsilon), \quad \text{for } T < h.$$

Restated, we have a time-log(space) tradeoff: $T \log(Space) \geq \log(Range)$, where $Space$ is the number of words in the pool of random words (exhibiting (hT)-wise independence) and $\log(Range)$ is the word size of the domain, range, and pool. Moreover, this lower bound and tradeoff applies to any algorithm with any level of precomputation, for we may simply view any internal storage and precomputed values as part of the pool measured by z.

The dependence on r, for r-practical schemes is more dramatic. Our constructions show that for any fixed r, linear hash functions can reduce the problem from a domain of size S to one of size n^{r+2}, provided the lookup table A contains random words from S.

We also remark that the counting argument for Theorem 4 gives an average case time bound. More precisely, let $T < h$ be the bound from Theorem 4. Then the time, averaged over all items in S, is at least $T - \frac{1}{|S|}(\frac{z^{\underline{T-1}}}{(h-2)^{\underline{T-2}}} + \frac{z^{\underline{T-2}}}{(h-2)^{\underline{T-3}}} + \ldots + z)$, which can be expressed as $T - O(1)$ when $z > ch$ for fixed $c > 1$.

4. Applications

The constructions of Section 3 show that (h)-wise independent hash functions, for non-constant $h < n^{\delta}$ and sufficiently small constant $\delta > 0$, can actually be programmed as constant time subroutines that require only a moderate size pool of random numbers as input. Thus we have established the computational feasibility of any probabilistic algorithm that has a performance bound based exclusively upon the use of such functions.

The two examples cited in this section are by no means self-contained. The first, which concerns the performance of double hashing, follows from an elaborate proof [16] based on $(O(\log n))$-wise independence. Consequently, Corollary 1 follows trivially. Yet even the performance bounds for full independence [7] are subtle and educational, and it is still not clear if the elegant proof technique of [7] can be translated into a proof for limited independence.

The second application, which concerns the pipelined emulation of an idealized $n \log n$ processor parallel machine on an n processor real machine, requires simple modifications of the original construction [13], which is based upon $(O(\log n))$-wise independence. The original algorithm is elegant but sufficiently elaborate that we only present the changes. In both applications, the original references are necessary and recommended for a complete understanding of the results.

Corollary 1. For fixed load factor $\alpha < 1$, $O(\log n)$-wise independent hash functions can be used for double hashing with constant expected probing for unsuccessful search.

∎

It should be noted that the [16] result only needs $O(\log n)$-wise independent hash functions that map, say, $[0, n^4] \mapsto [0, n-1]$.

Randomized routing schemes and PRAM emulation have had a substantial and fruitful recent literature [21], [17], [2], [12], [18], [19], [5], [13]. In particular, [5] and [13] show formally (and perhaps plausibly) how $n \log n$-processor Omega-like networks

can, with very high probability, emulate an $n \log n$-processor PRAM with an optimal performance penalty that is "only" a multiplicative factor of $\log n$.

Both Karlin and Upfal [5] and Ranade [13] presented schemes for an $n \log n$-processor emulation of $n \log n$-processor PRAM algorithms. The processors are interconnected on an $n \times \log n$ Omega network. For this configuration, no pipelining is possible with a model featuring 100% randomized memory references, because each PRAM emulation step causes the network to be effectively saturated for $O(\log n)$ time. Thus, their feasibility results, which were based on hash functions comprising $\log n$ degee polynomials sustain no performance penalty for evaluating such a polynomial for each memory reference. Given the $\log n$ performance cost for referencing, Karlin and Upfal did not need to address the much less significant issue of what to do about hashing collisions at the memory cell level; it simply cannot be a problem when $O(\log n)$ time is available to locate each item. Ranade [13] mentions the issue and shows that a scheme using $\log n$ reads per fetch readily solves the problem: his solution is to specify the location of items by their row number (which is in $[0, n-1]$) and the cell address of their module, but modulo the $\log n$ modules in a row of an $n \times \log n$ Omega network.

Now that highly independent hash functions can be evaluated in constant time, it is natural to reexamine these models to see if optimal speed-up can be achieved by pipelining these algorithms on machines that feature a reduced ratio of processor density to routing capacity. The idea of pipelining that exploits large scale parallel slackness to mask network latency can be traced to Smith [S-78], and has also been a subject of theoretical study in [11] and [20].

It is a simple matter to adapt the [5] and [13] constructions to emulate an $n \log n$ PRAM machine on a machine having one column of n processors interconnected by an $n \times \log n$ Omega network. A PRAM step of $n \log n$ parallel instructions is emulated by executing $\log n$ of the instructions in a pipeline of each processor.

The machine would also have n memory modules, say, one per processor. Ranade's

Common PRAM emulation scheme applies with a few simple modifications: First, the hash function would still be used to map the PRAM address $x \in [0, n^k - 1]$ into $[0, \log n - 1] \times [0, n-1] \times [0, n^{[k-1]}]$. The data packets are always kept locally lexicographically sorted (with the value x used to break and disambiguate ties). The first field, which, in Ranade's scheme, designated the column number of the destination module, is still used for the sorting of packets, but has no meaning in this case since only one column is active. The local process number (in $[1, \log n]$) for each packet might be explicitly listed in a separate field. The first phase of the algorithm requires each processor to provide its data in sorted order to the next stages as appropriate. This preprocessing can be done simply by following Ranade's approach: each processor uses its row of switches as a systolic bubble sorter for its packets. The "column" numbers cannot be arbitrarily set to the local process number for emulation of the Common PRAM, since combining would not be adequate to guarantee that only $O(\log n)$ messages would arrive at each memory module, per PRAM step. Such a scheme would, however, be adequate for an EREW emulator, and in this case the bubblesort can be skipped.

Now we can attain optimal speedup in emulation mode:

Corollary 2. A T-time $n \log n$ processor PRAM algorithm with n^k words of shared memory can, with high probability, be emulated on a pipelined n processor $n \times \log n$ Omega network in time $O(T \log n)$.

Proof: The only issue to address concerns memory contention at the cell and module levels.

We first observe that the n^k data are indeed well distributed among the n modules. From (h, μ)-wise independence, we have that the expectation of the h-th moments of the data counts apportioned among the modules is within a factor of μ of the fully random case: Formally, let $m_i(f)$ be the number of items, among the n^k data, that are mapped to module i by the hash function f, for $i = 1, 2, \ldots, n$. Then $\mathrm{E}_f[\binom{m_i}{h}] = \mathrm{E}_f[\sum_{\text{all size } h-\text{subsets of data}} \mathrm{Prob}\{\text{the subset hashes to module } i\}] \leq \binom{n^k}{h}\mu/n^h$. Hence

$\mathrm{E}[\binom{m_i}{h}] \leq \mu \frac{\binom{n^k}{h}}{n^h}$. We use this inequality in a Chebyshev bound: $\mathrm{Prob}\{m_i > \gamma n^{k-1} + h\} = \mathrm{Prob}\{\binom{m_i}{h} > \binom{\gamma n^{k-1}+h}{h}\} = \mathrm{Prob}\{\frac{\binom{m_i}{h}}{\binom{\gamma n^{k-1}+h}{h}} > 1\} \leq \mathrm{E}[\frac{\binom{m_i}{h}}{\binom{\gamma n^{k-1}+h}{h}}] \leq \mu \frac{\binom{n^k}{h}}{n^h \binom{\gamma n^{k-1}+h}{h}} < \frac{\mu}{\gamma^h}$. Thus $\mathrm{Prob}\{\max_i(m_i) > \gamma n^{k-1} + h\} < \frac{\mu n}{\gamma^h}$. Taking $h = O(\log n)$ sufficiently large and fixed $\gamma > 1$ gives a polynomially small probability† that γn^{k-1} items are hashed to any of the n modules.

We may suppose the aggregate storage capacity of the n modules is $(\gamma+1)n^k$ (multi-field) words. For simplicity, we may assume that the data is stored via bucket hashing with separate chaining, and that γ is large enough to accommodate this scheme trivially (say $\gamma = 3$). Thus the global address space of a module is size n^{k-1}, and each module can store γn^{k-1} chained elements in storage external to its formal table space.

In bucket hashing, an item hashed to a given location can be found in time proportional to the number of items hashed to the same address, since these colliding items are stored in a linked list. Thus the time required to satisfy r references to a single module is proportional to the sum of the list lengths for the r locations.

For the pipelined emulation of a single PRAM step, we can measure delays due to local collisions by a number of random variables, which get their randomness entirely from the hash function used to translate variable names into address locations. Let, for a single $\log n$ deep superstep that batches together *one* $n \log n$-way PRAM operation, n_i be the number of memory references to module i, for $i = 1, 2, \ldots, n$. Let l_i be the sum of the list lengths of the locations referenced in module i. Then the portion of the running time, for the single step, that is due to local processing within each module module is simply l_i, and the maximum of these random variables measures the intrinsic delay due to local retrievals. It is easy to use a buffer to sequence the return of data at times consistent with Ranade's original emulation algorithm, provided sufficient delay is introduced to ensure that all internal processing is successfully completed, with high

† A value is polynomially small if it depends on parameters that can be set so that it is less than $\frac{1}{n^c}$ for any fixed c and sufficiently large n.

probability. The delay has two parts, one due to routing, which is $O(\log n)$ in size with very high probability [13], and a second due to local access of multiple items hashed to module i.

We shall declare the pipelined emulation to fail at any step where $n_i \geq \beta \log n$, or $l_i \geq 4\beta \log n$. Now,

$$\text{Prob}\{\max_i(n_i) \geq \beta \log n \vee \max_i(l_i) \geq 4\beta \log n\} \leq n \times \text{Prob}\{n_1 \geq \beta \log n\} + n \times \text{Prob}\{n_1 < \beta \log n \wedge l_1 \geq 4\beta \log n\}.$$

Using (h, μ)-wise independence with $h \geq \beta \log n$, we calculate that among a batch of $\nu \leq n \log n$ memory references, the probability, that at least $\beta \log n$ of them will be hashed to (bucket) locations within memory module 1, is bounded by the expected number of such $(\beta \log n)$-subsets:

$$\text{Prob}\{n_1 \geq \beta \log n\} \leq \mu \frac{\binom{n \log n}{\beta log n}}{n^{\beta \log n}} < \frac{\mu}{(\beta \log n)!} <_s \frac{\mu}{(\frac{\beta}{e} \log n)^{\beta \log n}}.$$

This probability is superpolynomially small in n for fixed β.

We may use $(4\beta \log n, \mu)$-wise independence to overestimate $\text{Prob}\{l_1 \geq 4\beta \log n \wedge n_1 < \beta \log n\}$ very crudely as the expected number of pairs (S_1, S_2) where S_1 is a set of $j < \beta \log n$ references, among the actual $\nu \leq n \log n$ memory references, that hash onto module 1, and S_2 is a set of $3\beta \log n$ elements, among the $n^k - \nu$ unreferenced items, that hash into the hashing image of S_1.

$$\begin{aligned}
\text{Prob}\{l_1 \geq 4\beta \log n \wedge n_1 < \beta \log n\} &< \mu \sum_{j=1}^{\beta \log n - 1} \binom{n \log n}{j} (\frac{1}{n})^j \binom{n^k}{3\beta \log n} (\frac{j}{n^k})^{3\beta \log n} \\
&\leq \mu \sum_{j=1}^{\beta \log n - 1} \frac{(\log n)^j j^{3\beta \log n}}{j!(3\beta \log n)!} \\
&\leq \mu \frac{(\log n)^{\beta \log n} (\beta \log n)^{3\beta \log n}}{(\beta \log n)!(3\beta \log n)!} \\
&\leq_s \mu (\frac{e}{\beta})^{\beta \log n} (\frac{e}{3})^{3\beta \log n}.
\end{aligned}$$

This probability is polynomially small in n. Hence $\text{Prob}\{\max_i(n_i) \geq \beta \log n \vee \max_i(l_i) \geq 4\beta \log n\}$ is polynomially small.

Choosing suitably large constant β and $h = 4\beta \log n$ gives the desired performance bounds. ∎

Double hashing (c.f [16]) provides a formally simpler hashing method with essentially the same performance.

A formulation comparable to Corollary 2, conditioned on the existence of suitable hashing algorithms and corresponding hardware, was recently given in [20]. Versions of the basic counting estimates, with the exception of the bound on the aggregate number of collision items encountered by a batch of references queued at one memory module, can be found in [11] along with some early analysis of pipelining and various hashing schemes. It should also be noted that the Fast Fourier Transform can be used to evaluate k evaluations of a degree k polynomial in $k \log^2 k$ time (c.f. [1]). Thus it is possible to use the above pipeline strategy on n processors with $\log n$ degree hash functions to attain a performance cost of $(\log \log n)^2$ operations per memory reference rather than a naive $\log n$. We have shown that this multiplicative performance penalty can be reduced to $O(1)$.

5. Conclusions

Real machines have significant amounts of memory. We have shown how to exploit this capacity to store a sublinear sized database of random words in local memory to define highly independent hash functions that can be evaluated in constant time. For the development of probabilistic algorithms and the use of large scale parallel machines, this capability has, at least, theoretical importance. We have also shown that such functions have an intrinsic tradeoff between their evaluation time and the storage reserved for precomputed data (or their amortized evaluation time and the space reserved for active storage).

The high independence exhibited by our hash functions enriches the class of probabilistic algorithms that can be shown to achieve their expected performance in real computation. Proofs need not be restricted to h-wise independence for constant h, and probability estimates can use the probabilistic method to calculate the expected number of h-tuples satisfying various behavior criteria.

It is worth noting that the fast hash functions described in this paper are not really necessary for pure routing problems. After all, if an adequately random assignment of intermediate destinations provides, with very high probability, nearly optimal performance in a Valiant-Brebner style of routing [21], then the same destinations could be used for many consecutive routings.

What these fast hash functions really provide is nearly uniform mappings of data to modules and cell locations and a convenient way to assert that with high probability, no step in an n^k emulation sequence takes more than $O(\log n)$ time to complete. Thus, fast hash functions are even important for fast deterministic routing schemes, if large amounts of data have to be stored in a randomized manner. In addition, hash functions computed from destination addresses provide a way for common memory references to be fully combined en route in Ranade's simple queue management scheme, and this is important if combining is required to avoid hot spot contention.

From such a perspective, this work gives a theoretical foundation for the very pragmatic use of Memory Management Units. This paper gives a formal proof that such organizations work well in pipelined environments for a model of computation that is feasible and "only" a constant factor slower than methods used in practice.

From a more abstract perspective, we have exposed a very close equivalence between the true space-time computational complexity of (h)-wise independent hash functions and single instances of bipartite graphs on $[0, n-1] \times [0, n^\epsilon]$ that have low input degree d and have good expansion properties for small vertex sets. A spatially compact graph representation that can be used to compute the adjacency list of an input vertex in time $T_G = cd$ gives a time $T_f = T_G$ hash function with a high degree of independence, when augmented with a pool of n^ϵ random numbers. Similarly, a family of ϵd highly independent hash functions gives such a graph with $T_G = \epsilon d T_f$, albeit with an additive spatial cost of $\epsilon d n^\epsilon$ for the random numbers. It is worth remarking that the equivalence holds in this direction because our probability estimates in Section 2 were calculated

from h-way expectations, and never used full independence. The resource blowup is the modest factor ϵd because a random function value in $[0, n-1]$ gives $\frac{1}{\epsilon}$ points in $[0, n^{\epsilon}-1]$. A crude application of our lower bound imposes the requirement that $d > 1/\epsilon$, while our hash function construction gives sufficiency with $d = 2/\epsilon + 1$.

The most significant open question is how to find good weak expander-like graphs that are defined by short efficient programs. The discovery of such an object might have a very beneficial effect on the practicality of such a class of functions.

Acknowledgements

The author thanks J.P. Schmidt for stimulating discussion.

Appendix 1

Fact 1: Let $P_k = \{p \mid p \text{ is prime and } p \in (n^k \log m, (2+\beta)n^k \log m)\}$, for some small suitably fixed $\beta > 0$. Then

$$\forall x \neq y \in D: \ \text{Prob}_{p \in P_k}\{x = y \bmod p\} < n^{-k}.$$

Proof: [9],[4] By the Prime Number Theorem, $|P_k| = \frac{(1+\beta)n^k \log m}{k \log n + \log\log m}(1 - o(1))$, whence fewer than $1/n^k$ of the elements of P_k can divide $|x-y|$, since $\prod_{p \in P_k} p > (n^k \log m)^{|P_k|} > (m)^{n^k}$. ∎

Fact 2: Let $F_0(p) = \{h \mid h(x) = (ax + b \bmod p) \bmod n^k, \ a \neq 0, b \in [0, p-1]\}$, where $p > n^k$ is prime. Then

$$\forall x \neq y \in [0, p-1]: \ \text{Prob}_{f \in F_0(p)}\{f(x) = f(y)\} \leq n^{-k}.$$

Proof: [3] Given x and y, $x, y \in [0, p-1]$, $x \neq y$, the number of different $f \in F_0(p)$ where $f(x) = f(y)$, is precisely the number of 2×2 linear systems in a and b:

$$\begin{cases} ax + b = c + dn^k \bmod p \\ ay + b = c + en^k \bmod p \end{cases} \qquad c, d, e \geq 0; \ c + dn^k < p; \ c < n^k; \ e \neq d; \ c + en^k < p.$$

Now $c + dn^k$ can have p different values. The remaining parameter e cannot be set to d because this would give $a = 0$. Thus there are at most $\lceil p/n^k - 1 \rceil$ different values

available for e. Since there are $p(p-1)$ different functions in F_0, and $f(x) = f(y)$ for at most $p\lceil p/n^k - 1\rceil \leq p\frac{p-1}{n^k}$ of them, the result follows. ∎

Combining Facts 1 and 2 shows that a hash function selected at random from $F_0^k = \cup_{p\in P_k} F_0(p)$ will, with probability exceeding $1 - 2\binom{s}{2}n^{-k}$, map s items from D into $[0, n^k]$ with no collisions at all among its $\binom{s}{2}$ pairs. We could take $k = 4$, so that the probability of a collision is below $1/n^2$, and assume the functions $F_(h)$ are defined in (1) for $p \approx n^4$.

References

[1] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.

[2] N. Alon, Z. Galil, and V.D. Milman. Better Expanders and Superconcentrators. *Journal of Algorithms*, **8**, 1987, pp. 337–347.

[2] R. Aleliunas. Randomized parallel communication, 13*th PODC*, Aug., 1982, pp. 60-72.

[3] J.L. Carter and M.N. Wegman Universal Classes of Hash Functions, *Journal of Computer and System Sciences* 18, pp. 143–154 (1979).

[4] M.L. Fredman, J. Komlós and E. Szemerédi. Storing a Sparse Table with $O(1)$ Worst Case Access Time, *Journal of the Association for Computing Machinery*, Vol 31, No. 3, July 1984, pp. 538–544.

[5] A. Karlin and E. Upfal. Parallel Hashing - an Efficient Implementation of Shared Memory, 18*th Annual Symposium on Theory of Computing,* May, 1986, pp. 160–168.

[6] D.E. Knuth. *The Art of Computer Programming*, Vol. 3: Sorting and Searching, Addison-Wesley, Reading, Mass., 1973.

[7] G. Lueker and M. Molodowitch. More Analysis of Double Hashing, 20*th Annual Symposium on Theory of Computing*, May, 1988, pp. 354–359.

[8] A. Lubotzky, R. Phillips, and P. Sarnak. Ramanujan Graphs, *Combinatorica*, **8**(3), 1988 pp. 261–277.

[9] K. Mehlhorn. On the Program size of Perfect and Universal Hash functions, 23*rd Ann. Symp. on Foundations of Computer Science*, 1982, pp. 170–175.

[10] K. Mehlhorn. *Data Structures and Algorithms 1: Sorting and Searching*, Springer-Verlag, Berlin Heidelberg, 1984.

[11] K. Mehlhorn and U. Vishkin. Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories, *Acta Informatica*, 21, 1984, pp. 339–374.

[12] N. Pippenger. Parallel communication with limited buffers, 25*th Annual Symposium on Theory of Computing*, May, 1984, pp. 127–136.

[13] A.G. Ranade. How To Emulate Shared Memory, 28*th Annual Symposium on Foundations of Computer Science*, October 1987, pp. 185–194.

[14] B. Smith. A pipelined, shared resource MIMD computer, *Proceedings 1978 International Conference on Parallel Processing*, 1978, pp. 6–8.

[15] J. Spencer. *Ten Lectures on the Probabilistic Method*, SIAM, 1987.

[16] J.P. Schmidt and A. Siegel. The analysis of closed hashing under limited randomness, 22*nd Annual Symposium on Theory of Computing*, 1990, pp. 224–234.

[17] E. Upfal. Efficient schemes for parallel computation, 13*th PODC*, Aug., 1982, pp. 55-59.

[18] E. Upfal. A probabilistic relation between desirable and feasible models of parallel computation, 16*th Annual Symposium on Theory of Computing*, May, 1984, pp. 258–265.

[19] E. Upfal and A. Wigderson. How to share memory in a distributed system, 25*th Annual Symposium on Foundations of Computer Science*, October 1984, pp. 1701–180.

[20] L.G. Valiant. *General Purpose Parallel Parallel Architectures*, TR-07-89, Center for Research in Computing Technology, Harvard University, Cambridge, MA, 1989.

[21] L.G. Valiant and G.J. Brebner. Universal schemes for parallel communication, 13*th Annual Symposium on Theory of Computing*, May, 1981, pp. 263–277.

[22] M.N. Wegman and J.L. Carter. New Classes and Applications of Hash Functions, 20*th Annual Symposium on Foundations of Computer Science*, October 1979, pp. 175–182.

[23] A.C. Yao. Should Tables Be Sorted?, *Journal of the Association for Computing Machinery*, Vol 28, No. 3, July 1981, pp. 615–628.

[24] A.C. Yao. Uniform Hashing Is Optimal, *Journal of the Association for Computing Machinery*, Vol 32, No. 3, July, 1985, pp. 687–693.